2019-2020

Aprendizaje Automático para la predicción del cáncer de pulmón

Un enfoque práctico al mundo de la medicina

Pablo Martín García, Aaron Reboredo Vázquez

**Aprendizaje automático en la evaluación de tumores para el cáncer de pulmón**

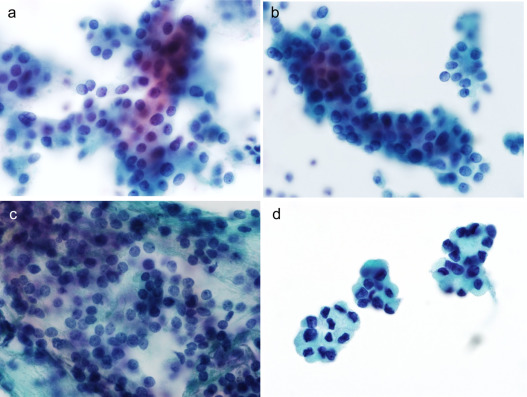
Esta memoria cubrirá las técnicas aplicadas a la hora de tratar el data set seleccionado para el proyecto final de la asignatura de Aprendizaje Automático.

**Motivación:**

“Cáncer” es una de las palabras más temidas hoy en día por el ser humano en el ámbito de la salud. Afortunadamente la ciencia y la medicina avanza y cada vez se están descubriendo formas menos abrasivas y más efectivas para combatirlo. Sin embargo, uno de los elementos cruciales para combatir la enfermedad es el rápido y preciso diagnóstico de la enfermedad para que pueda ser tratado con la mayor brevedad posible y con el tratamiento más adecuado. De esta manera nos resulta interesante el uso de algoritmos de aprendizaje automático que pueden ayudar a clasificar y a agilizar el proceso de diagnóstico de diversos tumores, en nuestro caso, el cáncer pulmonar.

El data set elegido contiene información relevante en la investigación del cáncer de pulmón para un estudio realizado en Wisconsin. En el data set se recogen los atributos de las células de una muestra de pulmón previamente analizadas digitalmente por imagen y les asocia un grado de Malignidad o Benignidad resultante de los análisis correspondientes.

El objetivo es comprobar que se puede generar un sistema capaz de predecir la malignidad o benignidad de un tumor pulmonar en función de los parámetros de entrada que se analizan bajo microscopio y solo utilizando el sistema de análisis por imagen con cierta efectividad. Como comentamos en este caso esas imágenes ya están procesadas y nosotros trabajaremos directamente con los datos extraídos de ese análisis por imagen.



**Imagen digitalizada de extracción masa pulmonar vía aspiración por aguja.**

**El data set:**

Nuestro data set está compuesto por 569 casos de análisis de los cuales 357 se corresponden con valores para los cuales el resultado es benigno y 212 se corresponden con casos malignos. De ellos conocemos 30 valores que se corresponderán con nuestras variables y que hacen referencia a la media, errores estándar y peores o mayores valores. De esta manera y cuando leamos el data set lo procesaremos para trabajarlo como una matriz de 569 x 30 con los valores de las variables y un vector resultado de dimensión 30 con 0 para el resultado benigno y 1 para el caso de haber obtenido un resultado maligno.

Nuestro set de datos se compone de:

Variables:

Los campos sobre los que vamos a realizar nuestro análisis o sobre los que vamos a ejecutar nuestros sistemas de predicción son los siguientes:

* Radio: Hace referencia a la distancia del punto central a un punto del perímetro del tumor.
* Textura: Hace referencia a la variedad de tonalidades de la imagen del tumor.
* Perímetro: Hace referencia a la longitud del borde del tumor o las aglutinaciones de este.
* Área: Hace referencia a la superficie que abarca el tumor.
* Suavidad de los bordes: Hacer referencia a la forma de los bordes del tumor Esto es la variación de longitudes de los radios.
* Compacidad: Es una medida a partir del perímetro del tumor y de la superficie que ocupa.
* Grado de simetría: Hace referencia a la equidad en las distancias de los radios a partir del eje perpendicular a los radios analizados.
* Concavidad: Hace referencia a la severidad de las porciones cóncavas del contorno del tumor.
* Cantidad de concavidades: Hace referencia al número total de concavidades visibles en la muestra.
* Dimensión fractal: Es una medida de la proximidad de las irregularidades de los bordes. Da una idea de la forma que tiene el perímetro del tumor.

Se analizan las medias de:

El radio, textura media de la mancha, perímetro medio, área, suavidad de los bordes, la compacidad, la concavidad, número de concavidades, grado de simetría, dimensión fractal.

Se analiza el error estándar en:

El radio, la textura, el perímetro, el área, la suavidad de los bordes, la compacidad, la concavidad, número de puntos cóncavos, simetría y dimensión fractal.

Se analizan también y se tienen en cuenta para el análisis los peores casos de (medidas más grandes y preocupantes dentro de las variables descritas):

Radio, textura, compacidad, perímetro, área, suavidad, textura, radio, concavidad, mayor cantidad de número de puntos cóncavos, peor simetría, y peor dimensión fractal.

Para trabajar con nuestra base de datos aplicaremos principalmente tres técnicas de análisis que nos permitirán predecir y clasificar los casos de análisis y comprobar mediante los valores de prueba (el data set), la fiabilidad o porcentaje de acierto de los procedimientos.

* Regresión logística
* Redes neuronales
* SVM

**Regresión lineal y gradiente descendiente:**

De esta manera empezaremos aplicando regresión lineal. A continuación, estudiaremos el caso mediante una regresión logística y terminaremos por implementar una red neuronal que nos permita discernir y clasificar los casos de estudio en función de los valores de entrada descritos anteriormente.

En el primer caso trabajaremos con una regresión lineal con varias variables, que se corresponden con cada una de las columnas de nuestro data set y con las descritas anteriormente. Haremos la aproximación por descenso de gradiente y comprobaremos tanto su similitud en las predicciones a los resultados obtenidos en la ecuación normal como su similitud con los valores reales para así poder extraer un porcentaje de acierto del análisis.

Es posible trabajar con este tipo de regresión puesto que tras un primer análisis o representación de las variables y su malignidad y benignidad nos hemos dado cuenta de que dos a dos existía una correlación que se podría aproximar a lo lineal entre las variables analizadas y el diagnóstico obtenido.

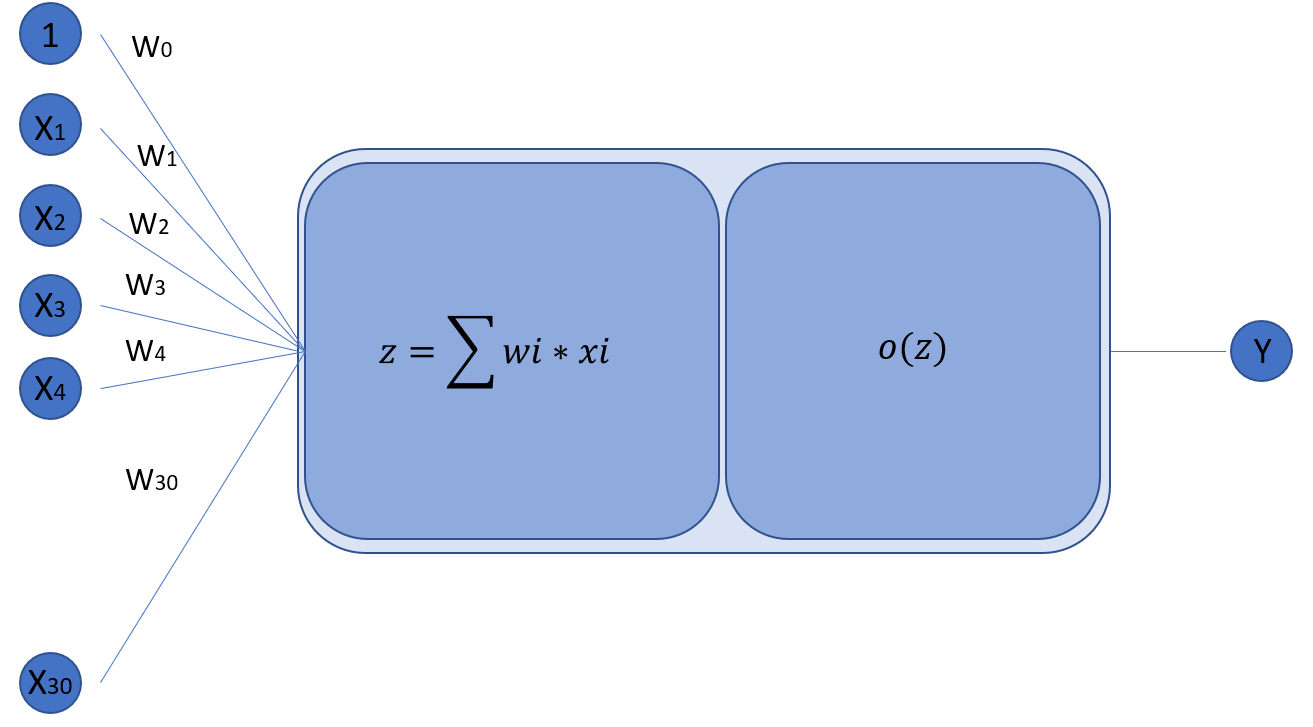
Los valores predichos por el procedimiento del gradiente descendiente se alejan de los valores reales que deberíamos obtener (recordemos que trabajábamos con 0 y con 1), pero que podemos modelizar o ajustar de tal manera que todos aquellos valores que no superen el 0.5 sean tomados como valor 0 (benignidad) y todos aquellos que superen el 0.5 sean considerados como 1(malignidad). Si practicamos este ajuste tanto a la predicción por gradiente descendiente como a la predicción normal obtenemos un 100 % de equivalencia entere ambas predicciones. Si comparamos los resultados predichos con los resultados reales obtenemos un 87% de coincidencia en las predicciones para el uso de regresión lineal mediante gradiente descendiente, lo que nos pone en una predicción no ideal, pero en cierta medida favorable.

Este modelo de predicción, sin embargo, no es el ideal, puesto que tenemos que modelizar los resultados obtenidos. La manera más eficaz para llevar un análisis de clasificación como el que queremos hacer tenemos otros modelos bastante más precisos y correctos como pueden ser la regresión logística, redes neuronales o las SVM, que nos brindan modelos más adecuados al tipo de resultado que queremos predecir (clasificación) y como veremos, más precisos.

**Regresión logística:**

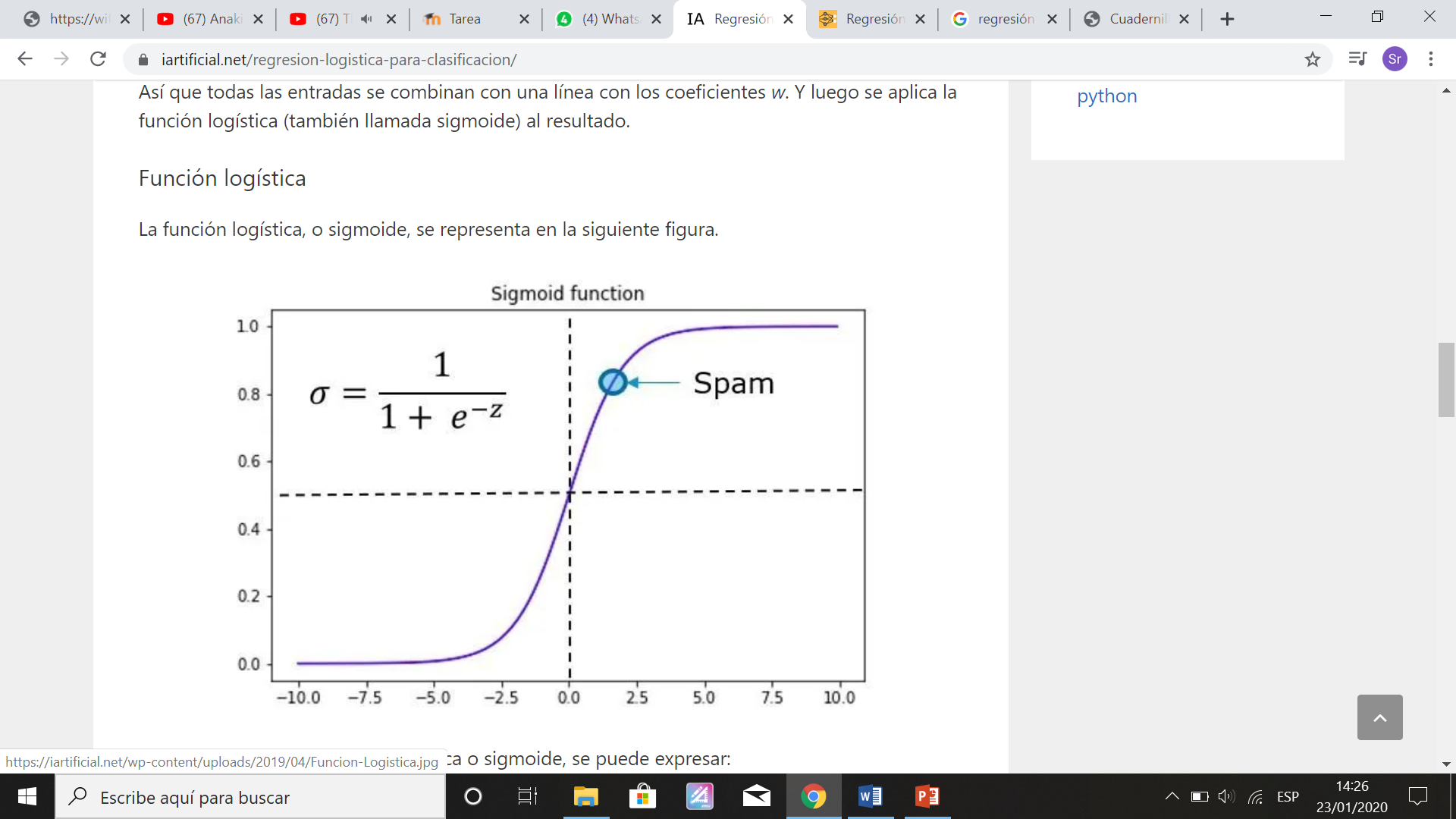
El primer paso lógico a la hora de lanzar nuestro análisis es empezar por el más sencillo de los tres que nos hemos planteado. Esta sencillez facilita y acelera la implementación de un sistema capaz de clasificar los valores de nuestro data set. Sin embargo, todo apunta a que esta sencillez nos hará pagar en precisión en la predicción.

La regresión logística es una técnica de aprendizaje automático relativamente sencilla. Es una de las técnicas más aplicadas en la industria debido a sus buenos resultados cuando se trabaja con muchos datos con interrelaciones sencillas. Además, el modelo de regresión logística nos permite aplicar la regularización, lo que nos permite generalizar y obtener mejores resultados.



Una regresión logística es en resumidas cuentas una pequeña red neuronal constituida por una sola neurona.

Cuando trabajamos con una regresión logística los datos se “preprocesan” y se combinan de manera lineal para posteriormente aplicarle la función logística y que en nuestro caso será la sigmoide.



Para la predicción de valores debemos pasar por el cálculo de los costes y los gradientes regularizados de trabajar con nuestra base de datos y procesarlos de tal manera que podamos obtener los parámetros óptimos, los pesos de nuestra pequeña red neuronal, que llamaremos thetas, para la regresión logística implementada. Esto pasa por hacer los cálculos de ambos y procesarlos.

El proceso del que hablamos es una minimización. Para ello existen diversos modelos de minimizado. Durante el curso hemos utilizado el modelo TNC, sin embargo, en nuestro caso hemos encontrado ciertos problemas a la hora de procesar nuestro data set, de tal manera que hemos decidido aplicar otro que nos ha brindado el cálculo de unos thetas o pesos óptimos para el problema que queremos resolver. El modelo elegido ha sido el “Nelder\_Mead. Hemos probado con otros modelos de minimización, pero los resultados obtenidos no han sido tan convincentes y se alcanzaba la excepción “divided by zero” demasiado pronto en el ciclo de cálculo de los pesos, obteniendo resultados no deseados.

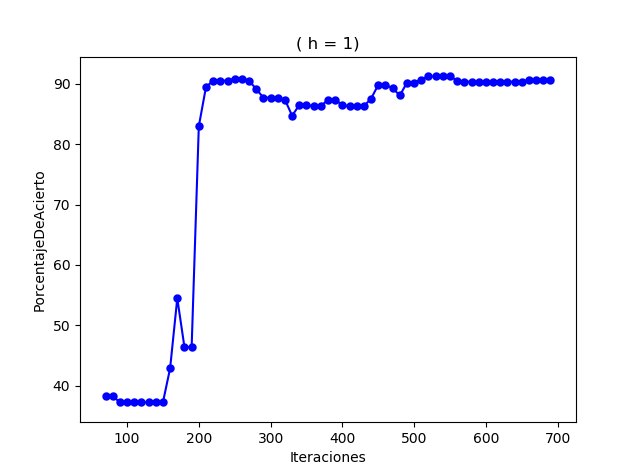
Este modelo tiene una peculiaridad que difiere del modelo TNC y es que se basa en el número de iteraciones para hacer los cálculos pertinentes a la hora de calcular las thetas óptimas para la predicción. De esta manera hemos creído conveniente experimentar por la afinidad de la predicción en función del número de iteraciones lanzadas para el algoritmo de minimización.

Como hemos mencionado anteriormente, nuestra regresión logística está regularizada, de tal manera que existe un parámetro de regularización que podría tener un gran impacto en la predicción de resultados, sin embargo, el modelo de minimización implementado y debido a la naturaleza de nuestro data set este tiene muy poco impacto en los resultados obtenidos. De esta manera lo hemos fijado a 1 y hemos hecho el análisis de eficacia en la predicción variando únicamente el número de iteraciones para el algoritmo de minimización.

Una vez tenemos los pesos óptimos podemos utilizarlos para calcular la predicción de nuestra regresión logística procesándolos con la función de normalización y posteriormente por la sigmoide.

Comparamos directamente el resultado con los valores reales

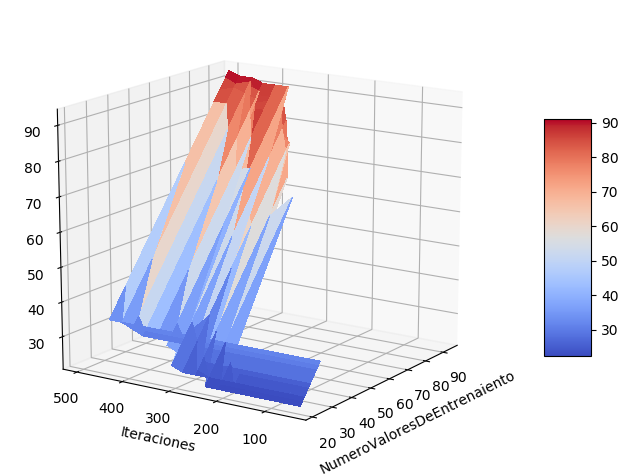
Los resultados obtenidos son los siguientes:



Podemos comprobar que para números bajos de iteraciones los pesos o thetas óptimos calculados nos llevan a unos porcentajes de aciertos muy bajos, pero a medida que aumentamos las iteraciones, encontramos un punto a partir del cual los porcentajes son bastante prometedores y estabilizándose llegados a las 500 iteraciones y brindándonos unos porcentajes de acierto en torno al 90%.

Sin embargo, en este caso, hemos utilizado todos los valores del dataset como valores de entrenamiento y como valores de verificación, donde lo verdaderamente interesante sería ver como de buena es nuestra regresión logística prediciendo valores nuevos a partir de una serie de valores de entrenamiento.

De esta manera hemos lanzado la regresión logística variando el número de iteraciones y el número de valores de entrenamiento, tomados de nuestro dataset, siendo los restantes, que no se han seleccionado para el entrenamiento, los valores de verificación.

Los resultados obtenidos los podemos representar en una gráfica 3D en el que se muestre el porcentaje de acierto de la red neuronal en función del número de iteraciones en la minimización y el número de valores de entrenamiento: 

Vemos como aumentando el número de iteraciones y el número de valores de entrenamiento aumenta el porcentaje de acierto, llegando a alcanzar valores al 90% como ocurría con anterioridad.

(Nótese que a la hora de representar la gráfica los valores mostrados para el número de valores de entrenamiento abarcan desde el 20 al 90. Sin embargo, estos números representan un rango que va desde los 50 valores de entrenamiento a los 500, esto puede ser debido a algún fallo a la hora de lanzar la representación de los ejes en 3D

**Redes Neuronales:**

Uno de los métodos o aproximaciones más utilizadas dentro del aprendizaje automático son las redes neuronales. Estas son una representación artificial mediante programación y código de una red neuronal humana.

Una red neuronal consta de una serie de capas importantes y sobre las que se realizan las operaciones correspondientes para llegar a la predicción buscada en función de las variables de entrada.

En nuestro caso la red neuronal constará de una capa de entrada, una capa oculta y una capa de salida.

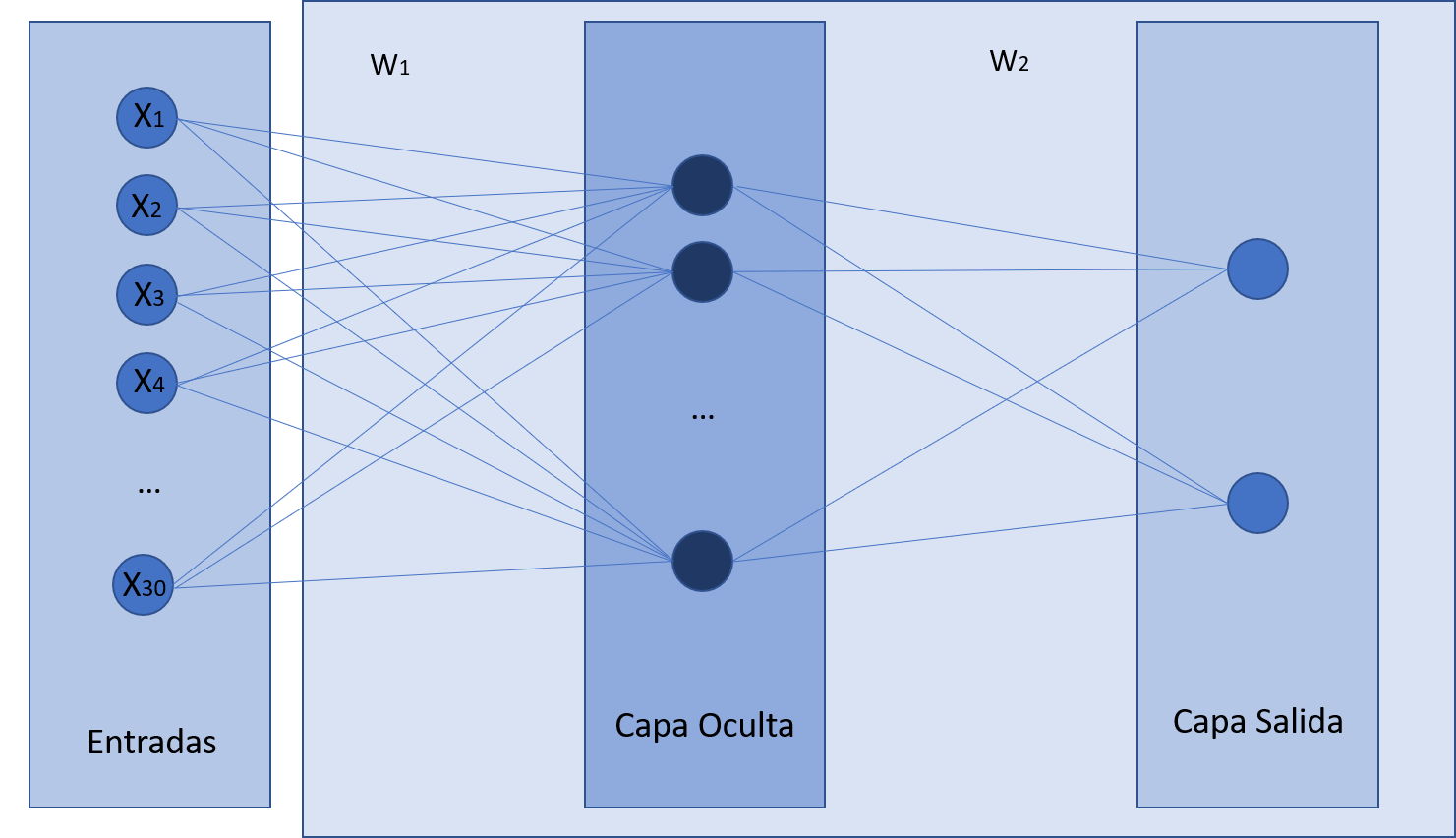
El tamaño de la capa de entrada se corresponde con el número de variables que construyen la predicción de nuestro caso y que se corresponde con el número de columnas de nuestra base de datos.

La teoría nos dice que basta con utilizar una sola capa oculta y que la gran mayoría de problemas prácticos son resolubles con una única capa de esta naturaleza.

En cuanto a la elección del número de neuronas en la capa intermedia haremos una primera aproximación ciñéndonos a la topología de la regla de la pirámide geométrica, esto es, el número de neuronas en la capa oculta se corresponde con la raíz cuadrada del número de neuronas en la salida veces el número de neuronas en la entrada (). Números muy altos incrementan mucho el tiempo de entrenamiento y números demasiado pequeños pueden ser causantes de “Underfitting”, es decir, que no se llegue a resolver el problema.

En cuanto al número de neuronas en la capa de salida haremos corresponder una por cada resultado que podemos obtener, es decir, dos (si el tumor es maligno (1) o benigno (0)).

Así, si tenemos 30 neuronas en la entrada y dos en la salida, la regla nos dice que el número de neuronas en la capa oculta puede rondar las 8, que será el número con el que trabajemos.



Implementación de la red Neuronal:

Cuando pensamos en trabajar con una red neuronal pensamos principalmente en dos fases claramente diferenciadas. La fase de propagación o “Forward propagation” y el cómputo de gradiente o “Backpropagation”.

La fase de propagación hace referencia al recorrido de “izquierda a derecha“, desde la entrada de la red neuronal a la salida para obtener los valores de activación que van emitiendo los resultados a las siguientes conexiones hasta obtener los valores finales.

La fase de Backpropagation es la siguiente a tener en cuenta, esta a su vez se divide en otras dos fases: una primera de propagación y una segunda de actualización de pesos (que son los parámetros que definen el estado y en qué medida van a ser activadas las neuronas adyacentes a una dada).

En esa primera fase de propagación se hace una primera pasada hacia adelante para generar las activaciones de salida de la red para posteriormente hacer una pasada hacia atrás desde las salidas para generar los deltas de todas las neuronas de la salida y de las pertenecientes a la capa oculta.

Como hemos comentado, la segunda parte trata sobre la actualización de los pesos.

El primer conjunto de pesos se asigna de manera aleatoria y es la propia Red Neuronal, mediante el algoritmo, la que “aprende” a ajustar el valor de los pesos para llegar al resultado o salida correcta.

Posteriormente iremos procesando los pesos hasta obtener los óptimos reduciéndose en el proceso el coste e iterando hasta llegar al mínimo, nuestro objetivo.

Repetiremos este procedimiento hasta que el resultado obtenido sea el satisfactorio.

Debemos tener en cuenta que trabajaremos bajo la influencia de una tasa de aprendizaje, como mencionábamos anteriormente, y que va a influir en la velocidad y calidad de la red. Por motivos prácticos y debido a que ya hemos trabajado con redes neuronales anteriormente, la tasa de aprendizaje con la que trabajaremos al principio será equivalente a la utilizada en redes previas y que será de valor 1.

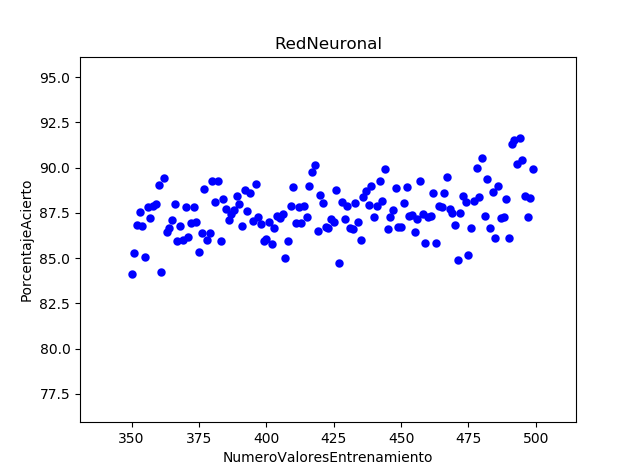
Trabajaremos por tanto con una red neuronal de 30 neuronas para la entrada (que se corresponde con el número de variables para tener en cuenta), con dos neuronas a la salida (que se corresponde con los resultados posibles, maligno o benigno) y con 8 neuronas en nuestra única capa oculta, siguiendo la regla de la pirámide.

Una vez aplicados los algoritmos pertinentes y tras haber estructurado la red neuronal hemos podido comprobar su efectividad. Hemos lanzado la red neuronal pasándole como parámetros de entrada todos nuestros casos de análisis para ver cuáles eran las perdiciones de nuestra red y el resultado lo hemos comparado con los datos reales.

El porcentaje de acierto para una tasa de aprendizaje de 1, 8 capas ocultas y el 100% del data set como valores de entrenamiento y como valores de análisis ronda en torno al 93%, que se corresponde con un resultado más favorable y preciso que el obtenido en algoritmos como el de descenso de gradiente y que nos harían pensar en el uso de una red neuronal como la programada para estudios como el que estamos llevando antes que cualquiera de los otros algoritmos utilizados.

Llegados a este punto sería interesante modificar ciertos parámetros de la red neuronal para ver cómo afecta a los resultados obtenidos y poder de esta manera elegir el mejor escenario para la predicción de valores.

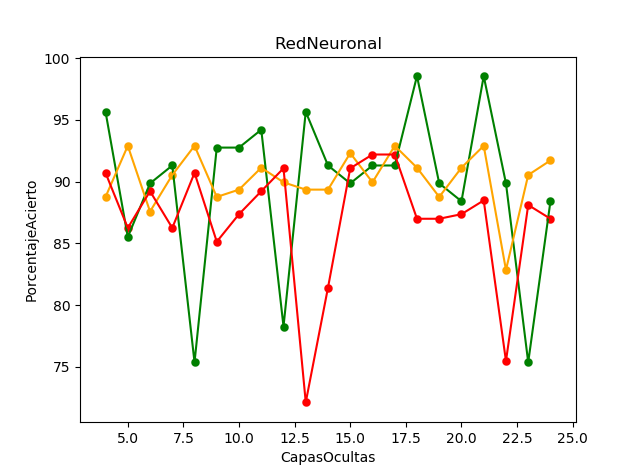
Un primer paso podría ser variar el número de valores de entrenamiento, y es que nuevamente el objetivo es entrenar la red neuronal con una serie de valores y posteriormente pasarle unos nuevos comprobando si predice adecuadamente los resultados. Hemos lanzado nuestra red neuronal variando el número de valores de entrenamiento utilizados de manera similar a como lo hemos hecho anteriormente, de tal manera que los valores de comprobación son los restantes que no hemos utilizado para entrenar la red neuronal:



Podemos ver como a medida que entrenamos más nuestra red neuronal (más valores de entrenamiento), esta nos da mejores resultados.

Teniendo esto en cuenta, el siguiente análisis lo haremos sobre una red entrenada con unos 450 valores de entrenamiento, utilizando los 110 restantes para comprobar la eficacia de nuestra red neuronal.

Nuestro primer análisis fue sobre una red que siguiendo la regla de la pirámide constaba de 8 capas ocultas. Lanzaremos ahora nuestra red neuronal variando el número total de capas ocultas y veremos cuál es el resultado obtenido. Hemos aprovechado también en este paso para comparar las posibles diferencias que existirían al variar el número de iteraciones para nuestro modelo de minimización, los resultados son los siguientes:

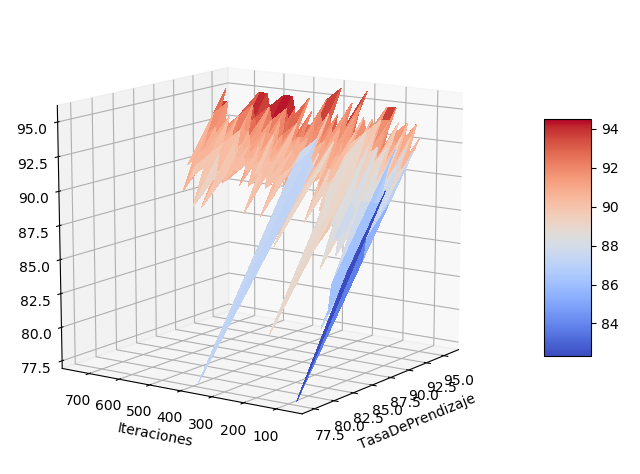


La gráfica verde, naranja y roja representan la variación en la precisión de acierto en función del número de capas ocultas de redes en las que se ha iterado un número de 700, 500 y 300 veces respectivamente a la hora de realizar la minimización para obtener los pesos óptimos.

Podemos comprobar que la gráfica verde predomina sobre las otras dos, indicándonos que a mayor número de iteraciones en nuestro método de cálculo de pesos óptimos obtendríamos mejores resultados, de la misma manera que los obtenemos al aumentar el número de capas ocultas. Si que es cierto que las gráficas también nos dicen que a medida que incrementamos mucho el número de iteraciones la dispersión en las predicciones obtenidas aumenta. Esto puede ser debido a que al aumentar el número de iteraciones nuestro modelo de minimización alcanza de manera más habitual situaciones de dividendo por 0 que pueden ser los que nos están dando esos porcentajes más bajos.

En un caso práctico eliminaríamos de la ecuación esas desviaciones aleatorias quedándonos con los valores más próximos al resto.

Por último, hemos decidido lanzar un estudio en el que se compruebe como afectan de manera conjunta el número de iteraciones y la tasa de aprendizaje



Podemos comprobar como a medida que aumenta el número de iteraciones y la tasa de aprendizaje ganamos precisión en la predicción, así como que la dispersión en los valores predichos se reduce, obteniéndose los valores más altos para las iteraciones más numerosas y nuestras tasas de aprendizaje más altas (analizado desde tasas de aprendizaje entre 0.2 y 5).

Utilizando los valores de iteraciones para la minimización, número de capas ocultas y número de valores de entrenamiento obtenemos unos porcentajes que alcanzan porcentajes en torno al 95%, superiores a los obtenidos anteriormente en nuestra regresión logística. Sin embargo, cabe resaltar que este incremento en la precisión se penaliza también con un incremento en los tiempos de ejecución y cálculo de resultados, que se ha hecho notable durante el desarrollo de la práctica y el entrenamiento de nuestras redes neuronales.

SVM:

Los SVM o máquinas de soporte vectorial son una serie de algoritmos de aprendizaje supervisado que están especialmente diseñados para resolver problemas de clasificación y regresión.

De la misma manera que hacíamos con nuestra red neuronal se debe utilizar una serie de valores para entrenarla y construir de esta manera un modelo capaz de predecir valores de la misma naturaleza que los que hemos utilizado como entrenamiento.

Será gracias a este modelo y a la aplicación de “funciones Kernel” como seremos capaces de elaborar un nuevo sistema para predecir resultados para nuestra base de datos.

La implementación a nivel de código es en este caso más sencilla, ya que la mayor parte de la algoritmia viene dada por la propia máquina de vectores.

Para empezar, seleccionaremos un conjunto de valores de entrenamiento como hacíamos anteriormente, 400 para el entrenamiento y los restantes para comprobar el acierto en las predicciones.

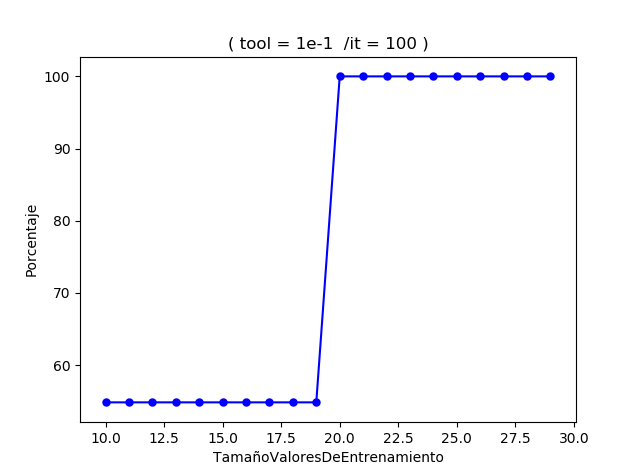
Con esos valores utilizamos los parámetros óptimos sigma y C de nuestra SVM y que a continuación pasaremos al modelo de entrenamiento para obtener la función gaussiana óptima que nos permitirá hacer las predicciones correspondientes.

El modelo de entrenamiento elegido es en nuestro caso el gaussiano que hace uso de la función kernel para realizar los cálculos correspondientes.

Después de obtener la función correspondiente podemos llamar directamente a su función de predicción a la que por medio del kernel gaussiano y pasándole los valores de comprobación nos devolverá un vector de predicciones de la naturaleza de los resultados o variable dependiente de nuestro data set.

El valor obtenido es del 100%, un resultado que supera con creces al obtenido en la red neuronal y que nos da una fiabilidad excelente.

Sin embargo, tras obtener este resultado nos hemos propuesto investigar hasta qué punto este sistema de predicción es eficaz y hemos querido comprobar el porcentaje de acierto dependiendo del número de valores de entrenamiento.

Los resultados son los siguientes:

Vemos como los resultados obtenidos son más que prometedores y es que basta con un número algo superior a los 20 valores de entrenamiento para obtener unas predicciones muy cercanas al 100%.

Esto convierte a las SVM en el mejor modelo para la predicción de valores de la naturaleza de nuestra base de datos. Si que es cierto y necesario recalcar que aquí es, aún si cabe, mucho más notable el coste en tiempo de cálculo frente a las Redes Neuronales y a la Regresión Logística, y es que estamos hablando de procesos que tardan entre los 2-5 minutos en mostrar resultados, que teniendo en cuenta la “simplicidad” de nuestro data set, son tiempos más que considerables.

**Conclusiones**

Con este proyecto hemos podido comprobar la versatilidad y el alcance que puede llegar a tener el aprendizaje automático y cómo es posible utilizar sus modelos, en nuestro caso de predicción, no solo en ámbitos como los de la informática sino en otros muchos como es el caso de la medicina.

Hemos visto cómo podemos atajar un mismo problema desde diferentes puntos de vista o utilizando implementaciones diferentes y como dependiendo de las mismas los resultados son más o menos favorables. Así mismo, para cada una de las implementaciones hemos podido comprobar como variando ciertos parámetros como el número de iteraciones en los procesos de minimización, las tasas de aprendizaje, el término de regularización o el número de valores utilizados para entrenamiento los porcentajes varían.

Los porcentajes obtenidos en los tres procedimientos principales son bastante altos y a medida que hemos ido utilizando sistemas más complejos hemos ido incrementando su valor, pasando de valores que no superaban el 90% en nuestra regresión logística hasta valores que rozan el 100% de acierto, pasando por números en torno al 94% para nuestra red neuronal.

Cabe destacar, por último, que este incremento en la eficacia de predicción ha supuesto un precio en tiempo de cálculo, puesto que procesos casi inmediatos como el de la regresión logística se han convertido en minutos de espera para el caso de nuestra SVM, con tiempos intermedios para los cálculos realizados por nuestra red neuronal.

**Apéndice:**

Código:

import numpy as np

import matplotlib.pyplot as plt

from matplotlib import cm

from mpl\_toolkits.mplot3d import Axes3D

import math

from math import e

from pandas.io.parsers import read\_csv

from sklearn.preprocessing import PolynomialFeatures

import scipy.optimize as opt

import random

from sklearn import svm

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def data\_csv(file\_name):

    "Takes the data from the csv file and tranfers it to a numpy array"

    values\_ = read\_csv(file\_name, header=None).values

    return values\_.astype(float)

def data\_builder(data):

    X = data[:, 2:]

    Y = data[:, 1]

    return X, Y

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def data\_visualization(X, Y):

    pos\_0 = np.where(Y == 0)

    pos\_1 = np.where(Y == 1)

    plt.xlabel('Radius')

    plt.ylabel('Perimeter')

    plt.scatter(X[pos\_0, 0], X[pos\_0, 3], marker = '+', c = "green")

    plt.scatter(X[pos\_1, 0], X[pos\_1, 3],  marker = '.',c = 'red')

    plt.show()

def vectors\_coincidence\_percentage(a, b):

    #Calcula el porcentaje de coincidencia dados dos vectores a, b

    coincidences\_array = a == b

    coincidences = sum(map(lambda coincidences\_array : coincidences\_array == True, coincidences\_array  ))

    percentage =100 \* coincidences/coincidences\_array.shape

    return percentage

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Regresión lineal, gradiente descendiente.

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Funcion de normalizado

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def normalize(X):

    X\_normalized = np.zeros((X.shape[0], X.shape[1]))

    mu = np.zeros(X.shape[1])

    sigma = np.zeros(X.shape[1])

    np.mean(X, axis = 0, out = mu)

    np.std(X, axis = 0, out = sigma)

    for i in range(X.shape[0]):

        for j in range(X.shape[1]):

            X\_normalized[i,j] = (X[i,j] - mu[j])/sigma[j]

    return X\_normalized, mu, sigma

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Funcion hipótesis

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def H\_Theta(X, Z): #Hipótesis del mocelo lineal vectorizada

    return np.dot(X, Z)

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Funcion de costes regresión lineal

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def cost\_function\_linear\_regression(X, Y, Theta): #funcion de costes vectorizada

    H = H\_Theta(X,Theta)

    Aux = (H-Y)\*\*2

    sumatory = Aux.sum()/(2 \* len(X))

    return sumatory

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Funcion normal, ajuste de la regresión lineal

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def normalEcuation(X,Y): #nos da los valores de theta que mejor se ajustan a nuestra regresión lineal

    #theta = (XT \* X)\*\*-1 \* XT \* Y

    XT = np.transpose(X)

    XT\_\_X = np.matmul(XT, X)

    XT\_\_X\_Inv = np.linalg.pinv(XT\_\_X, rcond = 10\*\*(-15))

    XT\_\_X\_Inv\_\_XT = np.matmul(XT\_\_X\_Inv, XT)

    thetas = np.matmul(XT\_\_X\_Inv\_\_XT, Y)

    return thetas

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Funcion descenso de gradiente

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def gradient\_descent(X, Y, alpha):

    m = X.shape[0]

    #construimos matriz Z

    th0 = 0.0

    th1 = 0.0

    th\_n = 0.0

    Z = np.zeros(X.shape[1]) #estamos tomando la dimensión de la X con la columna de 1, de tal manera que esta si coincide con el valor de ladimension que tiene que tener el vector de thetas

    Z\_ = np.zeros(X.shape[1] - 1)

    alpha\_m = (alpha/m)

    Thetas = np.array([Z]) #almacena los thetas que forman parte de la hipotesis h\_theta

    costs = np.array([]) #almacena los costes obtenidos durante el descenso de gradiente

    for i in range(1500):

        #Calculo de Theta 0

        #Sumatorio para el calculo de Theta0

        sum1 = H\_Theta(X, Z) - Y

        sum1\_ = sum1.sum()

        th0 -= alpha\_m \* sum1\_

        #Calculo Theta 1, 2, 3 ... n

        #Sumatorio para el calculo de Thetan

        for k in range(X.shape[1] - 1):

            sum2 =  (H\_Theta(X, Z) - Y) \* X[:, k + 1]

            sum2\_ = sum2.sum()

            th\_n -= alpha\_m \* sum2\_ #vamos calculando cada uno de los thn

            Z\_[k] = th\_n #almacenamos los thn calculados en un vector provisional

        #Actualizamos los nuevos thetas del vector Z

        Z[0] = th0

        for p in range(X.shape[1]-1):

            Z[p+1] = Z\_[p]

        Thetas = np.append(Thetas, [Z], axis= 0)

        #funcion de costes

        J = cost\_function\_linear\_regression(X,Y, Z)

        costs = np.append(costs, [J], axis = 0)

    return Thetas, costs

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Main Regresion Lineal, varias variables

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def regresion\_lineal\_multiples\_variables():

    cancer\_data = data\_csv("data.csv")

    X, Y = data\_builder(cancer\_data)

    X\_normalizada, mu, sigma = normalize(X)

    X = np.hstack([np.ones([np.shape(X)[0], 1]), X])

    X\_shape\_1 = np.shape(X\_normalizada)[0]

    X\_normalizada = np.hstack([np.ones([X\_shape\_1, 1]), X\_normalizada]) #le añadimos la columna de unos a la matriz ya normalizada

    Thetas, costs = gradient\_descent(X\_normalizada, Y, 0.0022) #los valores de theta aquí son los obtenidos normalizando la matriz, esto es, necesitamos "desnormalizarlos"

    Thetas\_normal\_Ecuation = normalEcuation(X, Y)

    normal\_prediction, gradient\_descent\_prediction = prediction\_vectors(X, Thetas\_normal\_Ecuation, Thetas, mu, sigma)

    normal\_prediction = vector\_prediction\_moddel(normal\_prediction)

    gradient\_descent\_prediction = vector\_prediction\_moddel(gradient\_descent\_prediction)

    print(vectors\_coincidence\_percentage(normal\_prediction, gradient\_descent\_prediction))#100 porciento de coincidencia con gradiente descendiente en comparación con distribución normal

    print(vectors\_coincidence\_percentage(gradient\_descent\_prediction, Y)) #86.99 porciento de acierto el la predicción con gradiente descendiente

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Cálculo vectores prediccion ecuación normal y descenso de gradiente

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def prediction\_vectors(X, Thetas\_normal\_Ecuation, Thetas, mu, sigma):

    shape\_thetas = np.shape(Thetas)[0]-1

    cancer\_samples = X.shape[0]

    normal\_predictions = np.zeros(cancer\_samples)

    gradient\_descendant\_predictions = normal\_predictions

    for i in range(cancer\_samples):

        cancer\_values = X[i, :]

        cancer\_Not\_normalize\_values = convert\_to\_list(cancer\_values)

        prediccion\_normal\_ecuation = H\_Theta(cancer\_Not\_normalize\_values ,Thetas\_normal\_Ecuation)

        normal\_predictions[i] = prediccion\_normal\_ecuation

        cancer\_normalize\_values = normalized\_test\_value(mu, sigma, cancer\_Not\_normalize\_values)

        prediccion\_gradiente\_descendiente = H\_Theta(cancer\_normalize\_values, Thetas[shape\_thetas])

        gradient\_descendant\_predictions[i] = prediccion\_gradiente\_descendiente

    return normal\_predictions, gradient\_descendant\_predictions

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Modelizado vectores predicción, a 0 todo lo que baja de 0.5 y a 1 todo lo que es mayor

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def vector\_prediction\_moddel(vector):

    moddel\_vector = np.zeros(vector.shape[0])

    moddel\_vector = (vector >= 0.5).astype(np.int)

    return moddel\_vector

#metodo auxiliar: convierte cada una de las filas o en nuestro caso caso de análisis en una lista para poder tratarlo posteriormente

def normalized\_test\_value(mu, sigma, row\_list) :

    dim = mu.shape[0]

    normalized\_values = []

    normalized\_values.append(1)

    for i in range(1, dim +1):

        normalized\_values.append((row\_list[0][i] - mu[i-1])/sigma[i-1])

    normalized\_values = [normalized\_values]

    return normalized\_values

#metodo auxiliar: convierte un vector a una lista

def convert\_to\_list(row):

    values\_list = []

    for i in range(row.shape[0]):

        values\_list.append(row[i])

    values\_list = [values\_list]

    return values\_list

#FIN DESCENSO DE GRADIENTE

#----------------------------------------------------------------------------------------------------------------------------------------------------------

#=======================================================================================

#Regresión logística

#=======================================================================================

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Función sigmoide

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def sigmoide\_function(X):

    e\_z = 1 / np.power(math.e, X) #np.power => First array elements raised to powers from second array, element-wise.

    sigmoide = 1/(1 + e\_z)

    return sigmoide

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Función hipotesis nula

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def H(X, Z): #Hipótesis del mocelo lineal vectorizada

    return np.dot(X, Z)

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Función de coste

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def cost(Thetas, X, Y):

    m = X.shape[0]

    #J(θ) = −(1/m) \* (A + B \* C)

    #J(θ) = −(1/m) \* ((log (g(Xθ)))T \* y + (log (1 − g(Xθ)))T \* (1 − y))

    #A

    X\_Teta = np.dot(X, Thetas)

    g\_X\_Thetas = sigmoide\_function(X\_Teta)

    log\_g\_X\_Thetas = np.log(g\_X\_Thetas)

    T\_log\_g\_X\_Thetas = np.transpose(log\_g\_X\_Thetas)

    y\_T\_log\_g\_X\_Thetas = np.dot(T\_log\_g\_X\_Thetas, Y)

    A = y\_T\_log\_g\_X\_Thetas

    #B

    one\_g\_X\_Thetas = 1 - g\_X\_Thetas

    log\_one\_g\_X\_Thetas = np.log(one\_g\_X\_Thetas)

    T\_log\_one\_g\_X\_Thetas = np.transpose(log\_one\_g\_X\_Thetas)

    B = T\_log\_one\_g\_X\_Thetas

    #C

    C = 1 - Y

    J = (-1/m) \* (A + (np.dot(B, C)))

    \_gradient = gradient(Thetas, X, Y)

    return J, \_gradient

def two\_power(X):

    return X\*\*2

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Función de coste regularizado

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def cost\_regularized(Thetas, X, Y, h):

    m = X.shape[0]

    Thetas\_ = Thetas

    #J(θ) = (cost(Thetas, X, Y)) + D

    #J(θ) = [−(1/m) \* ((log (g(Xθ)))T \* y + (log (1 − g(Xθ)))T \* (1 − y)))] + (λ/2m)\*E(Theta^2)

    cost\_ = cost(Thetas, X, Y)[0]

    #D

    Thetas\_ = two\_power(Thetas\_)

    D = h/(2\*m) \* np.sum(Thetas\_)

    J\_regularized = (cost\_) + D

    g = gradient\_regularized(Thetas, X, Y, h)

    return J\_regularized, g

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Función de gradiente

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def gradient(Thetas, X, Y):

    m = X.shape[0]

    #(δJ(θ)/δθj) =(1/m)\*XT\*(g(Xθ) − y)

    X\_Teta = np.dot(X, Thetas)

    g\_X\_Thetas = sigmoide\_function(X\_Teta)

    X\_T = np.transpose(X)

    gradient = (1/m)\*(np.dot(X\_T, g\_X\_Thetas - Y ))

    return gradient

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Función de gradiente regularizado

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def gradient\_regularized(Thetas, X, Y, h):

    m = X.shape[0]

    #(δJ(θ)/δθj) =(1/m)\*XT\*(g(Xθ) − y) + (λ/2m)(Theta)

    gradient\_ = gradient(Thetas, X, Y)

    g\_regularized = gradient\_ + (h/m)\*Thetas

    return g\_regularized

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Función thetas optimos

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def optimized\_parameters(Thetas, X, Y):

    result = opt.fmin\_tnc(func = cost, x0 = Thetas, fprime = gradient, args = (X, Y) )

    theta\_opt = result[0]

    return theta\_opt

def minimice\_rl(cost, Thetas, X, Y):

    #Calcula los parámetros optimos de pesos para nuestra red neuronal

    fmin = opt.minimize(fun=cost, x0=Thetas, args=(X, Y),

    method='Nelder-Mead', jac=True, options={'maxiter': 550})

    result = fmin.x

    return result

def minimice\_rl\_r(cost, Thetas, X, Y, h, iter):

    #Calcula los parámetros optimos de pesos para nuestra red neuronal

    fmin = opt.minimize(fun=cost\_regularized, x0=Thetas, args=(X, Y, h),

    method='Nelder-Mead', jac=True, options={'maxiter': iter})

    result = fmin.x

    return result

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Función thetas optimos regularizados

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def optimized\_parameters\_regularized(Thetas, X, Y, h):

    result = opt.fmin\_tnc(func = cost\_regularized, x0 = Thetas, fprime = gradient\_regularized, args = (X, Y, h) )

    theta\_opt = result[0]

    return theta\_opt

def logistic\_regresion\_evaluation(X, Y, Z):

    H\_ = H(X, Z)

    H\_sigmoid = sigmoide\_function(H\_)

    H\_sigmoid\_evaluated = (H\_sigmoid >= 0.5).astype(np.float)

    return vectors\_coincidence\_percentage( Y ,H\_sigmoid\_evaluated)

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Función Regresión logística

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def logistic\_regresion():

    cancer\_data = data\_csv("data.csv")

    X\_, Y\_ = data\_builder(cancer\_data)

    X\_V = X\_

    X\_m = np.shape(X\_)[0]

    X\_V = np.hstack([np.ones([X\_m,1]),X\_])

    Thetas = np.zeros(X\_V.shape[1])

    cost\_ = cost(Thetas, X\_V, Y\_)[0]

    gradient\_ = gradient(Thetas, X\_V, Y\_)

    print(cost\_)

    print(gradient\_)

    optimized\_thetas = minimice\_rl(cost, Thetas, X\_V, Y\_)

    print("Thetas optimos: ", optimized\_thetas)

    cost\_ = cost(optimized\_thetas, X\_V, Y\_)[0]

    print("Coste optimo : ", cost\_)

    percentage\_correct\_clasification = logistic\_regresion\_evaluation(X\_V, Y\_, optimized\_thetas)

    print("Porcentaje de acierto : ", percentage\_correct\_clasification)

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Función Regresión logística regularizada

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def regularized\_logistic\_regresion(h , iter): #90.6% obtenido de coincidencia, a medida que hemos ido incrementando el número de iteraciones hemos ido consiguiendo un mayor porcentaje de coincidencia

    cancer\_data = data\_csv("data.csv")

    X\_, Y\_ = data\_builder(cancer\_data)

    X\_poly = X\_[:h, :]

    Y\_\_ = Y\_[:h]

    Thetas = np.zeros(X\_poly.shape[1])

    cost\_r = cost\_regularized(Thetas, X\_poly, Y\_\_, h)[0]

    gradient\_r = gradient\_regularized(Thetas, X\_poly, Y\_\_, 1)

    optimized\_thetas\_regularized = minimice\_rl\_r(cost\_regularized, Thetas, X\_poly, Y\_\_, 1, iter)

    percentage\_correct\_clasification = logistic\_regresion\_evaluation(X\_[h:, :], Y\_[h:], optimized\_thetas\_regularized)

    #print("Porcentaje de acierto : ", percentage\_correct\_clasification)

    return percentage\_correct\_clasification

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Funciones pintado para análisis regresión logística

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def triDi\_regresion():

    h = np.arange(50, 500, 28)

    i = np.arange(50,500,28)

    X,Y = np.meshgrid(h, i)

    Z = regularized\_logistic\_regresion(X, Y)

    fig = plt.figure()

    ax = Axes3D(fig)

    surf = ax.plot\_surface(X, Y, Z, cmap= cm.coolwarm, linewidths= 0, antialiaseds = False)

    fig.colorbar(surf, shrink = 0.5, aspect = 5)

    plt.xlabel("NumeroValoresDeEntrenaiento")

    plt.ylabel("Iteraciones")

    plt.show()

def regularized\_logistic\_regresion\_preparation(X, Y):

    z = X

    for i in range(X.shape[0]):

        for j in range(X.shape[1]):

            z[i, j] = regularized\_logistic\_regresion(X[i,j], Y[i,j])

    return z

def pintaFunciontasaRegresionIteraciones(\_from = 70, to = 700, step = 10, Xlabel = "Iteraciones", Ylabel = "PorcentajeDeAcierto", name = "( h = 1)"):

    x = np.arange(\_from, to, step)

    s = x.shape[0]

    y = np.zeros(s)

    for i in range(s):

        y[i] = regularized\_logistic\_regresion(1,x[i])

    plt.plot(x, y, color = "blue")

    plt.scatter(x, y, color = "blue", linewidths= 0.05)

    plt.xlabel(Xlabel)

    plt.ylabel(Ylabel)

    plt.title(name)

    plt.show()

#FIN Regresión logística

#----------------------------------------------------------------------------------------------------------------------------------------------------------

#=======================================================================================

#Redes Neuronales

#=======================================================================================

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Función auxiliar Despliega vector

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Nos permite desplegar en un vector otros dos

def unrollVect(a, b):

    thetaVec\_ = np.concatenate((np.ravel(a), np.ravel(b)))

    return thetaVec\_

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Función auxiliar Despliega vector

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Nos permite plegar en un vector otros dos

def rollVector(params, num\_entradas, num\_ocultas, num\_etiquetas):

    vector1 = np.matrix(np.reshape(params[:num\_ocultas \* (num\_entradas + 1)], (num\_ocultas, (num\_entradas + 1))))

    vector2 = np.matrix(np.reshape(params[num\_ocultas \* (num\_entradas + 1):], (num\_etiquetas, (num\_ocultas + 1))))

    return vector1, vector2

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Función pesos aleatorios

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Nos permite inicializar dos vectores de pesos aleatorios para cada una de las capas de la red neuronal con entrada L\_in y salida L\_out

def generate\_Random\_Weights(L\_in, L\_out):

    e\_ini = math.sqrt(6)/math.sqrt(L\_in + L\_out)

    e\_ini= 0.12

    weights = np.zeros((L\_out, 1 + L\_in))

    for i in range(L\_out):

        for j in range(1 + L\_in):

            rnd = random.uniform(-e\_ini, e\_ini)

            weights[i,j] = rnd

    return weights

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Funciones sigmoide para la red neuronal

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def sigmoid(x):

    return 1/(1 + np.exp((-x)))

def sigmoid\_Derived(x):

    return x \* (1 - x)

def sigmoid\_Gradient(z):

    sig\_z = sigmoid(z)

    return np.multiply(sig\_z, (1 - sig\_z))

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Funciones procesado Y, preparación para la red neuroal

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Devuelve la salida en forma de matriz lista para ser utilizada por nuestros métodos de la red neuronal

def y\_onehot(y, X, num\_etiquetas):

    m = X.shape[0]

    y\_onehot = np.zeros((m, num\_etiquetas))

    for i in range(m):

        y\_onehot[i][int(y[i])] = 1

    return y\_onehot

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Función optimización de pesos

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Calcula los parámetros optimos de pesos para nuestra red neuronal

def minimice(backprop, params, num\_entradas, num\_ocultas, num\_etiquetas, X, y, tasa\_aprendizaje, iter ):

    fmin = opt.minimize(fun=backprop, x0=params, args=(num\_entradas, num\_ocultas, num\_etiquetas, X, y, tasa\_aprendizaje),

    method='TNC', jac=True, options={'maxiter': iter})

    result = fmin.x

    return result

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Función pasada hacia adelante

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Método pasada hacia adelante para la implementación de la red neuronal

#Nos devuelve los parámetros de activación de la red neuronal y el valor h

def forward(X, theta1, theta2):

    m = X.shape[0]

    a1 = np.insert(X, 0, values=np.ones(m), axis=1)

    z2 = np.dot(a1, theta1.T)

    a2 = np.hstack([np.ones([m, 1]), sigmoid(z2)])

    z3 = np.dot(a2, theta2.T)

    h = sigmoid(z3)

    return a1, z2, a2, z3, h

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Función Coste Red Neuronal

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Funcion que calcula el coste base(sin regularizar)

def cost\_n\_r(params, num\_entradas, num\_ocultas, num\_etiquetas, X, y, tasa\_aprendizaje):

    m = X.shape[0]

    X = np.matrix(X)

    y = np.matrix(y)

    theta1 = np.matrix(np.reshape(params[:num\_ocultas \* (num\_entradas + 1)], (num\_ocultas, (num\_entradas + 1))))

    theta2 = np.matrix(np.reshape(params[num\_ocultas \* (num\_entradas + 1):], (num\_etiquetas, (num\_ocultas + 1))))

    h = forward(X, theta1, theta2)[4]

    J = (np.multiply(-y, np.log(h)) - np.multiply((1 - y), np.log(1 - h))).sum() / m

    return J, theta1, theta2

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Función Coste Regularizado Red Neuronal

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Cálculo del coste con el ajuste de regularización

def cost\_Regularized\_n\_r(params, num\_entradas, num\_ocultas, num\_etiquetas, X, y, tasa\_aprendizaje):

    m = X.shape[0]

    J\_, theta1, theta2 = cost\_n\_r(params, num\_entradas, num\_ocultas, num\_etiquetas, X, y, tasa\_aprendizaje)

    J\_regularized =  J\_ + (float(tasa\_aprendizaje) /

            (2 \* m)) \* (np.sum(np.power(theta1[:, 1:], 2)) + np.sum(np.power(theta2[:, 1:], 2)))

    return J\_regularized

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Función Cálculo de gradientes

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Calculo de los deltas del gradiente (no regularizado)

def backProp\_Deltas(a1, z2, a2, z3, h, theta1, theta2, y, m):

    delta1 = np.zeros(theta1.shape)

    delta2 = np.zeros(theta2.shape)

    d3 = h - y

    z2 = np.insert(z2, 0, values=np.ones(1), axis=1)

    d2 = np.multiply((theta2.T \* d3.T).T, sigmoid\_Gradient(z2))

    delta1 += (d2[:, 1:]).T \* a1

    delta2 += d3.T \* a2

    delta1 = delta1 / m

    delta2 = delta2 / m

    return delta1, delta2

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Función Cálculo de gradientes regularizados

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def backProp\_Deltas\_regularized(a1, z2, a2, z3, h, theta1, theta2, y, m, tasa\_aprendizaje):

    delta1, delta2 = backProp\_Deltas(a1, z2, a2, z3, h, theta1, theta2, y, m)

    delta1[:, 1:] = delta1[:, 1:] + (theta1[:, 1:] \* tasa\_aprendizaje) / m

    delta2[:, 1:] = delta2[:, 1:] + (theta2[:, 1:] \* tasa\_aprendizaje) / m

    return delta1, delta2

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Función paso hacia atrás

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Pasada hacia adelante y hacia atrás en nuestra red neuronal, nos calcula el gradiente y el coste correspondientes a nuestra red neuronal

def backprop(params, num\_entradas, num\_ocultas, num\_etiquetas, X, y, tasa\_aprendizaje, regularize = True):

    m = X.shape[0]

    X = np.matrix(X)

    y = np.matrix(y)

    theta1 = np.matrix(np.reshape(params[:num\_ocultas \* (num\_entradas + 1)], (num\_ocultas, (num\_entradas + 1))))

    theta2 = np.matrix(np.reshape(params[num\_ocultas \* (num\_entradas + 1):], (num\_etiquetas, (num\_ocultas + 1))))

    a1, z2, a2, z3, h = forward(X, theta1, theta2)

    if regularize:

        delta1, delta2 = backProp\_Deltas\_regularized(a1, z2, a2, z3, h, theta1, theta2, y, m, tasa\_aprendizaje)

    else :

        delta1, delta2 = backProp\_Deltas(a1, z2, a2, z3, h, theta1, theta2, y, m)

    J = cost\_Regularized\_n\_r(params, num\_entradas, num\_ocultas, num\_etiquetas, X, y, tasa\_aprendizaje)

    grad = unrollVect(delta1, delta2)

    return J, grad

def neuronal\_prediction\_vector(sigmoids\_matrix) :

    #calcula los valores predichos por la red neuronal dada una matriz de sigmoides

    # Será generada por la funcón forward.

    samples = sigmoids\_matrix.shape[0]

    y = np.zeros(samples)

    for i in range(samples):

        y[i] = np.argmax(sigmoids\_matrix[i, :])

    return y

def neuronal\_succes\_percentage(X, y, weights1, weights2) :

    #Compara los valores predichos por la red neuronal para unos

    sigmoids\_matrix = forward(X, weights1, weights2)[4]

    y\_ = neuronal\_prediction\_vector(sigmoids\_matrix)

    percentage = vectors\_coincidence\_percentage(y\_, y)

    return percentage

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Main Red Neurnal

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def neuronal\_red\_main(ocultas, num\_valores\_entrenamiento, iteraciones, tasa\_a):

    cancer\_data = data\_csv("data.csv")

    X, y = data\_builder(cancer\_data)

    tasa\_aprendizaje = tasa\_a

    num\_etiquetas = 2 #num\_etiquetas = num\_salidas. Maligno y Benigno

    num\_entradas = 30

    num\_ocultas = ocultas

    theta1 = generate\_Random\_Weights(num\_entradas, num\_ocultas)

    theta2 = generate\_Random\_Weights(num\_ocultas, num\_etiquetas)

    params\_rn = unrollVect(theta1, theta2)

    y\_ = y\_onehot(y, X, num\_etiquetas)

    params\_optimiced = minimice(backprop, params\_rn, num\_entradas, num\_ocultas, num\_etiquetas, X[:num\_valores\_entrenamiento, :], y\_[:num\_valores\_entrenamiento], tasa\_a, iteraciones)

    theta1\_optimiced, theta2\_optimiced = rollVector(params\_optimiced, num\_entradas, num\_ocultas, num\_etiquetas)

    percentage = neuronal\_succes\_percentage(X[num\_valores\_entrenamiento:], y[num\_valores\_entrenamiento:], theta1\_optimiced, theta2\_optimiced)

    #print("Percentage neuronal red : ", percentage) #hemos llegado a obtener hasta un 93.84% de acierto (oscilan entre el 87 y el 93 % de acierto)

    return percentage

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Funciones de pintado para el análisis de nuestra red neuronal

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def pintaFuncionCapasOcultasDependiente(\_from = 3, to = 25, step = 1, Xlabel = "CapasOcultas", Ylabel = "Porcentaje", name = "( v\_e = 400  /it = 700 / t\_a = 0.02)"):

    x = np.arange(\_from, to, step)

    s = x.shape[0]

    y = np.zeros(s)

    for i in range(s):

        y[i] = neuronal\_red\_main(x[i], 500, 700, 0.02)

    plt.plot(x, y, color = "green")

    plt.scatter(x, y, color = "green", linewidths= 0.05)

    plt.xlabel(Xlabel)

    plt.ylabel(Ylabel)

    plt.title(name)

    plt.show()

def pintaFuncionIteracionesDependiente(\_from = 10, to = 700, step = 10, Xlabel = "NumeroDeIteraciones", Ylabel = "PorcentajeDeAcierto", name = "( v\_e = 400  / c\_o = 8 / t\_a = 0.02)"):

    x = np.arange(\_from, to, step)

    s = x.shape[0]

    y = np.zeros(s)

    for i in range(s):

        y[i] = neuronal\_red\_main(8, 500, x[i], 0.02)

    plt.plot(x, y, color = "green")

    plt.scatter(x, y, color = "green", linewidths= 0.05)

    plt.xlabel(Xlabel)

    plt.ylabel(Ylabel)

    plt.title(name)

    plt.show()

def pintaFunciontasaDeAprendizajeDependiente(\_from = 0.05, to = 10, step = 0.05, Xlabel = "TasaDeaprendizaje", Ylabel = "PorcentajeDeAcierto", name = "( v\_e = 400  / c\_o = 8 / it = 700)"):

    x = np.arange(\_from, to, step)

    s = x.shape[0]

    y = np.zeros(s)

    for i in range(s):

        y[i] = neuronal\_red\_main(8, 500, 700, x[i])

    plt.plot(x, y, color = "green")

    plt.scatter(x, y, color = "green", linewidths= 0.05)

    plt.xlabel(Xlabel)

    plt.ylabel(Ylabel)

    plt.title(name)

    plt.show()

def pintaFuncionPorcentajeVsNumeroValoresEntrenamiento(\_from, to, step, function, Xlabel, Ylabel, name):

    x = np.arange(\_from, to, step)

    s = x.shape[0]

    y = np.zeros(s)

    for i in range(s):

        y[i] = neuronal\_red\_main(x[i], 500, 700, 0.02)

    plt.plot(x, y, color = "green")

    plt.scatter(x, y, color = "green", linewidths= 0.05)

    for i in range(s):

        y[i] = neuronal\_red\_main(x[i], 400, 700, 0.02)

    plt.plot(x, y, color = "orange")

    plt.scatter(x, y, color = "orange", linewidths= 0.05)

    for i in range(s):

        y[i] = neuronal\_red\_main(x[i], 300, 700, 0.02)

    plt.plot(x, y, color = "red")

    plt.scatter(x, y, color = "red", linewidths= 0.05)

    plt.xlabel(Xlabel)

    plt.ylabel(Ylabel)

    plt.title(name)

    plt.show()

def tri\_Di\_Red\_Neournal\_tasaAprendizajeVSIteranciones():

    h = np.arange(0.2, 5.2, 0.2)

    i = np.arange(70,770,28)

    X,Y = np.meshgrid(h, i)

    Z = neuronalRedPreparation(X, Y)

    fig = plt.figure()

    ax = Axes3D(fig)

    surf = ax.plot\_surface(X, Y, Z, cmap= cm.coolwarm, linewidths= 0, antialiaseds = False)

    fig.colorbar(surf, shrink = 0.5, aspect = 5)

    plt.xlabel("TasaDePrendizaje")

    plt.ylabel("Iteraciones")

    plt.show()

def neuronalRedPreparation(X, Y):

    z = X

    for i in range(X.shape[0]):

        for j in range(X.shape[1]):

            z[i, j] = neuronal\_red\_main(8,  450, Y[i,j], X[i,j])

    return z

#FIN Red Neouronal

#----------------------------------------------------------------------------------------------------------------------------------------------------------

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Funciones calculo Kernel Gausiano

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def gaussian\_Kernel(X1, X2, sigma):

    Gram = np.zeros((X1.shape[0], X2.shape[0]))

    for i, x1 in enumerate(X1):

        for j, x2 in enumerate(X2):

            x1 = x1.ravel()

            x2 = x2.ravel()

            Gram[i, j] = np.exp(-np.sum(np.square(x1 - x2)) / (2 \* (sigma\*\*2)))

    return Gram

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Función principal entrenamiento SVM

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def SVM\_gaussian\_training(X, y, c\_param, tol, max\_i, sigma):

    svm\_ = svm.SVC(C = c\_param, kernel="precomputed", tol = tol, max\_iter = max\_i)

    return svm\_.fit(gaussian\_Kernel(X, X, sigma=sigma), y)

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Funcion cálculo parámetros óptimos

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def optimal\_C\_sigma\_Parameters(X, y\_r, Xval, yval, max\_i, tool ):

    predictions = dict() #almacenaremos la infrmacion relevante en un diccionario

    for C in [0.01, 0.03, 0.1, 0.3, 1, 3, 10, 30]:

        for sigma in[0.01, 0.03, 0.1, 0.3, 1, 3, 10, 30]:

            model = SVM\_gaussian\_training(X, y\_r, C, tool, max\_i, sigma )

            prediction = model.predict(gaussian\_Kernel( Xval, X, sigma))

            predictions[(C, sigma)] = np.mean((prediction != yval).astype(int))

    C, sigma = min(predictions, key=predictions.get)

    return C, sigma

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Lógica principal SVM

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def SVM\_main(test\_values, X, y):

    #Parte 1.3

    tool = 1e-3

    iterations = 100

    y\_r = np.ravel(y)

    Xval = X[:test\_values, :]

    yval = y[:test\_values]

    optC, optSigma = optimal\_C\_sigma\_Parameters(X, y\_r, Xval, yval, iterations, tool)

    svm\_function\_optimal\_C\_sigma = SVM\_gaussian\_training(X, y\_r, optC, tool, iterations, optSigma)

    prediction = svm\_function\_optimal\_C\_sigma.predict(gaussian\_Kernel( X, X, optSigma))

    return vectors\_coincidence\_percentage(prediction, y)

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#Funciones de pintado análisis SVM

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def pintaSVM\_ValoresEntrenamientoVSPorcentaje(\_from = 10, to = 30, step = 1, Xlabel = "TamañoValoresDeEntrenamiento", Ylabel = "Porcentaje", name = "( tool = 1e-1  /it = 100 )"):

    cancer\_data = data\_csv("data.csv")

    \_X, \_y = data\_builder(cancer\_data)

    x = np.arange(\_from, to, step)

    s = x.shape[0]

    y = np.zeros(s)

    for i in range(s):

        y[i] = SVM\_main(x[i], \_X, \_y)

    plt.plot(x, y, color = "blue")

    plt.scatter(x, y, color = "blue", linewidths= 0.05)

    plt.xlabel(Xlabel)

    plt.ylabel(Ylabel)

    plt.title(name)

    plt.show()